2 Recall: Traversal by Pointer

```
int const SIZE = 5;
int arr[SIZE] = { 1, 2, 3, 4, 5 };
```

- Traversal by Pointer
  - Walk a **pointer** across the array elements.
  - When you want an element, just dereference the pointer!

```
int *end = arr + SIZE;
for (int *ptr = arr; ptr != end; ++ptr) {
  cout << *ptr << endl;
}
```

Notice that "end" is really "one past the end"

Pointer starts at beginning of the array.

Continue until pointer at end.

Increment pointer.

Dereference pointer to current element.

6/6/2022

①通过 Pointer Traversal 的 interface 长什么样.

3 Can we use this for a std::vector?

```
vector<int> v = { 1, 2, 3, 4, 5 };
```

这里是行不通的，因为v是一个object 而不是一个address

- Let's try it…
  - What parts don't work?

```
int *end = v + SIZE;
for (int *ptr = v; ptr != end; ++ptr) {
  cout << *ptr << endl;
}
```

vector elts

class vector { T * begin() { return elts; } }

6/6/2022

① vector 照搬 arr 的 coding 为什么会出现问题

4 Can we use this for a std::vector?

```
vector<int> v = { 1, 2, 3, 4, 5 };
```

- Ask the container for the endpoints!
  - begin() and end() member functions

```
for (int *ptr = v.begin(); ptr != v.end(); ++ptr) {
  cout << *ptr << endl;
}
```

① Traversal 在 vector 中的正确用法.
② 因为 vector 是 continuous memory，所以可以"++"

5 Can we use this for a Linked List?

```
List<int> list;
// Assume we add 1, 2, 3, 4, 5 to the list
```

于是我们决定不使用普通的int pointer而是使用特殊的pointer：iterator。通过对operator的重载，实现pass by pointer。

- Let's try it…
  - What parts don't work?

list first last

Iterator

```
for (int *ptr = list.begin(); ptr != list.end(); ++ptr) {
  cout << *ptr << endl;
}
```

special pointer, nullptr, datum, overload

因为不是连续的，而使用ptr = ptr->next又会破坏pass by pointer

6/6/2022

① List 使用 vector 的 Traversal 的 interface 不行的原因.

6 Iterating Through a List

- Here's one way to do it…

```
int main() {
  List<int> list;
  int arr[3] = { 1, 2, 3 };
  fillFromArray(list, arr, 3);

  for (List<int>::Node *np = list.first; np != nullptr; np = np->next) {
    cout << np->datum << endl; // print each element
  }
}
```

- Problems:
  - This breaks the interface of the List. Nodes are an implementation detail we don't want to mess with here.
  - The Node type is private, so this won't even compile.

6/6/2022

① 使用之前 iterating through list 的方法的问题.

7 The Iterator Interface

- Iterators provide a common interface for iteration.
  - A generalized version of traversal by pointer.
  - An iterator "points" to an element in a container and can be "incremented" to move to the next element.
- Iterators[1] support these operations:
  - Dereference – access the current element. `*it`
  - Increment – move forward to the next element. `++it`
  - Equality – check if two iterators point to the same place. `it1 == it2` `it1 != it2`

[1] There are many different kinds of iterators. These operations are specifically required for input iterators.

6/6/2022

① iterator 的意义
② iterator 要重载的4个 operator.

8 Traversal by Iterator

- The big picture:
  - Walk an **iterator** across the elements.
  - When you want an element, just dereference the iterator!
  - We'll look at how to get the beginning and end iterators in just a bit…

Notice that "end" is really "one past the end"

```
Iterator end = list.end();
for (Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

Iterator starts at beginning of the container.

Continue until iterator at end.

Increment iterator.

Dereference iterator to current element.

6/6/2022

① Traversal by Iterator 的 interface.
② 将之前 vector 的"int *"替换成"Iterator."

5 Traversal by Iterator

- The big picture:
  - Walk an **iterator** across the elements.
  - When you want an element, just dereference the iterator!
  - We'll look at how to get the beginning and end iterators in just a bit…

```
Iterator end = list.end();
for (Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

① begin() 指的是哪里.
② end() 指的是哪里

Building an Iterator for a Linked List

6 What is an Iterator?

通过在iterator的内部重载operator使得使用者可以直接像操作pointer一样，而我们将这些operator重新定义来完成他们的目标。

- An iterator is an object that "works like a pointer".
- This can be implemented by a class that overloads the appropriate operators (*, ++, ==, !=).

```
class Iterator {
public:
  ___ & operator*() const;

  Iterator & operator++();

  bool operator==(Iterator rhs) const;

  bool operator!=(Iterator rhs) const;
  ...
};
```

Element type omitted for now.

Dereference – access the current element.

Increment – move forward to the next element.

Equality – check if two iterators point to the same place.

We'll discuss each of these in more detail in the coming slides… 3/27/2021

①"*"访问 node 中的 datum
②"++"让 iterator 指向下一个 node
③"=="&"!=" check 两个 iterator 是否指向同一个 node.

8 List Iterator: Data Representation

- How do we represent an end iterator?
  - "One past the end"
  - Use a null pointer as a sentinel value.

iterator中有一个member variable是node的pointer，当iterator中的pointer变成nullptr的时候就表示已经访问到了list的最后一个node。

Iterator: node_ptr 0

IntList: first, last; Node: datum 1, next; Node: datum 2, next; Node: datum 3, next; Node: datum 4, next 0

3/27/2021

①如何表示 end iterator

9 Implementing a List Iterator

List<int>::Iterator

1.因为iterator的兼容性很差，适用于set的不适用于vector，所以我们将iterator的定义放置在list当中，强调了他们之前的紧密联系。
2.同时作为nest class，在list的scope中，iterator这个class是可被访问的。
3.iterator class也可以使用list class上面所定义的template，这也是为什么要将他写成nest class的原因，保持template type相同。
4.iterator的class可以访问list中的private的members.

- Data representation
  - Store a pointer to the node holding the current element.
- The Iterator class is defined inside the List class.
  - This gives Iterator access to the private section of List, including the Node struct.
  - Iterator can also use the same template parameter T so that the * operator returns the correct element type.
  - Each different instantiation of the List template will have its own corresponding Iterator.

```
template <typename T>
class List {
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  ...
  class Iterator {
  public:
    T & operator*() const;
    ...
  private:
    Node *node_ptr;
  };
  ...
};
```

①为什么 Iterator 要定义在 List 的 public 中.(有什么好处)
②共用一个 typename T.
③ Iterator public function 有啥.
④ Iterator private member 有啥.

10 List Iterator: The * operator

```
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Returns the element this iterator points to.
template <typename T>
T & List<T>::Iterator::operator*() const {
  assert(node_ptr);
  return node_ptr->datum;
}
```

确认没有指向超出list范围的东西，不然就引用是undefined behavior

We return by reference to allow both reading and writing through an Iterator.

An iterator is dereferenceable if it points to some element in the container.

3/27/2021

① assert() 防止解引用 nullptr
② return type 要可读可写 (T&)
③ 返回 datum.

11 List Iterator: The ++ operator (prefix[1])

为了实现这种功能，我们必须return iterator本身，这样才能重复++操作

```
// REQUIRES: this is a dereferenceable iterator
// EFFECTS:  Increments this iterator to point to the next
//           element. Returns this iterator by reference.
template <typename T>
typename List<T>::Iterator & List<T>::Iterator::operator++() {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

The typename keyword is required here.

[1] The postfix increment operator can also be overridden. 3/27/2021

①"++"返回值支持重复++."&"
② Iterator 初始化要注明所属 List
③ List 若使用 Template 则要在最开头写 typename.
④"::"表明所属关系.

12 The typename Keyword

class x { }; x * y

- The typename keyword is required when naming a type nested inside another type that depends on a template parameter.

compiler会在这里疑惑，这到底是一个variable，还是一个type？
就比如这个例子：x*y
compiler不知道是一个variable叫x乘以另一个variable叫y，还是一个type为X的variable y的初始化。
所以在这里我们就很明确的告诉compiler这是一个type的名称，而非是一个variable的名称。

```
template <typename T>
void func() {
  IntList::Iterator it1;

  List<int>::Iterator it2;

  typename List<T>::Iterator it3;
}
```

No typename required, since IntList does not depend on the parameter T.

No typename required, since List<int> does not depend on the parameter T.

typename required, since List<T> depends on the parameter T.

dependent name

3/27/2021

①不用 template 不用
②用 template 但表明 type <int> 不用.
③用 template 且不表明 type <T> 用
④ compiler 不知其为 typename 还是 variable name.

13 List Iterator: The == operator[1]

```
// EFFECTS: Returns whether this and rhs are pointing to
//          the same place.
// NOTE:    The result is only meaningful if both are
//          pointing into the same underlying container.
template <typename T>
bool List<T>::Iterator::operator==(Iterator rhs) const {
  return node_ptr == rhs.node_ptr;
}
```

"=="号左边的地址直接通过iterator的private member得到，我们只需要在parameter中表明右侧的变量即可。

通过比较地址来确定他们所指的是不是同一个node。

pass-by-value，因为rhs所储存的信息很少，所以不会因为copy而浪费很长的时间

[1] The != operator is defined analogously. 3/27/2021

① Parameter 直接 pass-by-value 即可.
②比较其是否指向同一个 node.

14 Creating Iterators

- We'll provide two constructors for Iterator.

```
class Iterator {
public:
  // Public constructor. Creates an end Iterator
  Iterator()
    : node_ptr(nullptr) { }
  ...

private:
  // Private constructor. Creates an Iterator pointing
  // to the specified Node.
  Iterator(Node *np)
    : node_ptr(np) { }

  Node *node_ptr;
};
```

end, past-the-end, specific

指向list最后一个node的后面一位。nullptr表明list已经结束。

There's no need for the outside world to use this one. Node itself is private, after all.

因为node_ptr本身private，所以这constructor也是private。

这样做的意义是：main( ) { List <int>::Iterator it; it = list.begin() }

我们在创建之初不想立即初始化，所以让 default constructor 为 nullptr.

① Iterator 两种 constructor - public - private.
② public 的 default ctor 使用情境.
③ private 的 ctor 为啥 private.

15 Getting Iterators for a Container

- The missing piece from earlier was how to get the iterators for a container…

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);

List<int>::Iterator end = _____;
for (List<int>::Iterator it = _____; it != end; ++it) {
  cout << *it << endl;
}
```

How do we get beginning and end iterators?

- We'll implement begin() and end() functions for the List class that construct these iterators for us.

①如何用 Iterator 对 Container 进行遍历.

16 begin() and end()

我们希望通过两个方程得到list的start和end。
begin () 使用需要parameter输入node pointer的constructor。
end () 直接使用default constructor，因为他会直接指向一个nullptr代表着list的结束。
但是现在就有一个问题：在list public中定义的function无法访问nest class iterator中的private的东西。

```
template <typename T>
class List {
public:
  ...
  class Iterator {
  public:
    Iterator() : node_ptr(nullptr) { }
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }

  Iterator end() { return Iterator(); }
  ...
private:
  Node *first;
  ...
};
```

Clicker Question 18.1

What's wrong with this code?

A) Memory Leak
B) Missing const
C) Use of first
D) Use of ctor

The begin() function uses the constructor to create an iterator pointing to the first element.

这里无法compile，会报错

The end() function uses the default constructor to create and return a "past the end" iterator.

① List public function: begin() : end()

17 Friend Declarations

```
template <typename T>
class List {
public:
  ...
  class Iterator {
    friend class List;
  public:
    Iterator() : node_ptr(nullptr) { }
    ...
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }

  Iterator end() { return Iterator(); }
  ...
private:
  Node *first;
  ...
};
```

在 iterator 中就行

We use a friend declaration to give List special privileges to access the private members of Iterator.

List member functions, like begin(), can now access the private constructor.

It's easy to get this backwards. Remember that "friendship is given, not taken."

①如何让 List 中的 public member function 使用 Iterator 中的 private ctor.

18 Traversal by Iterator

- We now have all the pieces to implement and use the traversal by iterator pattern.

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);

List<int>::Iterator end = list.end();
for (List<int>::Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

Ask the List for iterators that define the sequence of elements.

① Iterator traversal 的 interface.

20 Iterator Big Three?

- Do we need custom versions of the Big Three for the Iterator class? Let's do an exercise…

Iterator: node_ptr; IntList: first, last; Node: datum 1, next; Node: datum 2, next; Node: datum 3, next; Node: datum 4, next 0

① Iterator 不须要 big three.

21 Exercise

Walkthrough: Iterator Big Three??

```
int main() {
  List<int> list;
  // Add to list so it contains 1, 2, 3
  List<int>::Iterator it1 = list.begin();
  ++it1;
  List<int>::Iterator it2 = it1;
  ++it2;
  // Draw memory at this point
}
```

我们发现让两个iterator指向同一个node，在对其中一个iterator进行某些操作并不会造成像之前的memory leak的问题。
这是因为：
1. iterator和pointer一样，只是指向list中的node，并不对list中的node进行具体的修改。
2. 当出了main function的scope之后，it1和it2都会被释放，并使用default的destructor来进行释放，这是正确的。
3.最终，当call list destructor时，所有的node都会被free。

① Iterator 不需要 Big three 的原因.

22 Exercise

```
int main() {
  List<int> list;
  // Add to list so it contains 1, 2, 3
  List<int>::Iterator it1 = list.begin();
  ++it1;
  List<int>::Iterator it2 = it1;
  ++it2;
  // Draw memory at this point
}
```

Clicker Question 18.2

If we add the dtor below:

A) Memory Leak
B) Use a dead object
C) Double Free
D) Dereference a null pointer
E) No errors occur

```
class Iterator {
  friend class List;
public:
  Iterator() : node_ptr(nullptr) { }
  ~Iterator() { delete node_ptr; } // Should we add this???
  ...
private:
  Iterator(Node *np) : node_ptr(np) { }
  Node *node_ptr;
};
```

如果我们这么做了，那么iterator清理一遍，list在清理一遍就会出现double delete的问题。
因为：iterator只是指向但不真正owning的address and data，所以deep copy和destructor delete的事情将由list来完成

①若 Iterator 使用 custom dtor，会发生什么.

23 The Iterator Interface

- Iterators provide a common interface for traversing a sequence of elements.
- They allow us to reuse the same code to work with many different kinds of containers as long as they provide an iterator interface.
- The STL containers work this way. For example:

```
vector<int> vec;
// Fill vec with numbers

vector<int>::iterator end = vec.end();
for (vector<int>::iterator it = vec.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

① Iterator interface 的优点...

24 Iterators Generic Functions

- A key strength of iterators is that we can write functions to work with iterators, rather than with a particular container.
- This allows the same function to be used with many different containers!
- The STL contains many functions, like `std::sort`, that work this way.

```
int main() {
  vector<int> vec; // fill with numbers
  sort(vec.begin(), vec.end());
}
```

① 一个 work with iterator 的 functor 可兼容不同 container.

25 Example: max_element

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {
  Iter_type maxIt = begin;
  for (Iter_type it = begin; it != end; ++it) {
    if (*it > *maxIt) {
      maxIt = it;
    }
  }
  return maxIt;
}

int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;
}
```

Start by assuming first element is the max.

Use traversal by iterator to check each element.

If we find a larger element, update maxIt to point to it.

Dereference returned iterator to get the element itself.

vector<int>::Iterator, 也可能是 list<int>::Iterator

① 兼容 function 的例子

26 Using max_element

- As long as we are working with a container that supports iterators, we don't ever have to write that maximum-finding loop again!

```
int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;

  List<int> list; // fill with numbers
  cout << *max_element(list.begin(), list.end()) << endl;

  List<Card> cards; // fill with Cards
  cout << *max_element(cards.begin(), cards.end()) << endl;

  int const SIZE = 10;
  double arr[SIZE]; // fill with numbers
  cout << *max_element(arr, arr + SIZE) << endl;
}
```

Pointers also work as iterators!

② 不同 container 使用相同 function.

27 Exercise

Clicker Question 18.3
Which of these is a correct generic `length` function?

```
template <typename Iter_type>
int length(Iter_type begin,
           Iter_type end) {
  int count = 0;
  List<int>::iterator it = begin;
  while(it != end) {
    ++count;
    ++it;
  }
  return count;
}
```

A

```
template <typename Iter_type>
int length(Iter_type begin, Iter_type end) {
  int count = 0;
  for(Iter_type it = begin; it < end; ++it) {
    ++count;
  }
  return count;
}
```

B

```
template <typename Iter_type>
int length(Iter_type begin, Iter_type end) {
  int count = 0;
  while(begin != end) {
    ++count;
    ++begin;
  }
  return count;
}
```

C

```
template <typename Iter_type>
int length(Iter_type begin, Iter_type end) {
  return end - begin;
}
```

D

```
int main() { // EXAMPLE
  List<int> list; // assume it's filled with some numbers
  cout << length(list.begin(), list.end()) << endl;
}
```

30 Iterator Invalidation

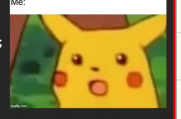
dangling pointer

```
int main() {
  List<int> list;
  list.push_back(1);
  list.push_back(2);

  List<int>::Iterator it = list.begin();
  List<int>::Iterator it2 = list.begin();
  cout << *it << endl; // OK
  cout << *it2 << endl; // OK

  list.erase(it);

  cout << *it << endl; // EXPLODE
  cout << *it2 << endl; // ALSO EXPLODE
}
```

Me: I wonder what happens if I delete the element this iterator is pointing to

Iterator: *segfault*

Me:



31 Iterator Invalidation

- Invalidated iterators are like dangling pointers – it's no longer safe to dereference them and try to access the object they point to.
- Seemingly innocuous operations on a container can result in iterator invalidation.
  - For example, iterators pointing into a vector are invalidated if an operation causes a grow.
- A function's documentation should specify which iterators, if any, it may invalidate.

Class List{

public:

class Iterator{

public: ++; *; ==; !=; constructor

private:

node * node_ptr;

constructor

}

private:

struct node{

};

node * first

node * last

};